Control-flow Guided Clause Generation for Property Directed Reachability

Xian Li and Klaus Schneider
Embedded Systems Group
Department of Computer Science
University of Kaiserslautern, Germany
http://es.cs.uni-kl.de

Abstract—Property directed reachability (PDR) has been introduced as a very efficient verification method for synchronous hardware circuits which is based on induction rather than fix-point computation. The method incrementally refines a sequence of clause sets that over-approximate the states that are reachable in finitely many steps. Even being valid, safety properties may not be provable by induction due to so-called counterexamples to induction (CTIs) that result from the over-approximation of the reachable states. Crucial steps of the PDR method therefore consist of (1) deciding about the reachability of states derived from counterexamples, and (2) generalizing them to clauses that cover as many unreachable states as possible that are then excluded from consideration by adding the generated clause to the reachable state approximation sequence.

In this paper, we describe a refinement of the PDR method for synchronous programs that makes effective use of the distinction between the control- and dataflow of synchronous programs. If a CTI candidate is found, we reduce it to its control-flow part and check whether the obtained control-flow states are unreachable in the corresponding extended finite state machine of the program. If so, we can immediately exclude all these states by adding the negation of the control-flow part as a new clause to the current reachable state approximations; otherwise, the usual steps of the PDR method are applied. This additional step in the PDR method is not expensive, and can significantly increase the performance of PDR.

I. INTRODUCTION

Property directed reachability (PDR) [1–5] is an inductive incremental algorithm originally introduced with the tool IC3 in [6] for model checking hardware circuits. It is currently considered as one of the most efficient methods for verifying safety properties as it often outperforms classic verification tools that have been optimized over many years. PDR avoids unrolling a system’s transition relation as well as fixpoint iterations, e.g., to compute the reachable states. Instead, it maintains a sequence of clause sets \( \Psi_0, \ldots, \Psi_k \) that over-approximate the states that are reachable in 0, \ldots, \( k \) steps, respectively, and that are all subsets of the states that satisfy the considered safety property \( \Phi \). PDR then incrementally proceeds in one of two ways: It either extends the sequence \( \Psi_0, \ldots, \Psi_k \) with a new clause set \( \Psi_{k+1} \) that covers the states reachable in at least \( k + 1 \) steps, or it narrows the clause sets \( \Psi_i \) by adding a new clause to them that was proved to contain only unreachable states.

Working with over-approximations \( \Psi_0, \ldots, \Psi_k \) instead of the precise sets of states that can be reached within \( i \) steps must be viewed as an advantage of the method, since it has this way the chance to prove a safety property more efficiently than methods that precisely compute the reachable states. On the other hand, the approximations have to be tight enough to make the induction work: Even though a safety property \( \Phi \) may hold on all reachable states, the induction step may still fail since there may exist unreachable states that satisfy \( \Phi \) while some of their successor states do not satisfy \( \Phi \). SAT/SMT solvers may therefore generate counterexamples for the induction step that may be just pseudo-counterexamples, also called counterexamples to induction (CTIs) in this context, so that PDR has to check whether given states can be reached within the first \( k \) steps using the clause sets \( \Psi_0, \ldots, \Psi_k \). If such a counterexample turns out to be just a CTI, it is generalized to a smaller clause that covers more unreachable states that are then excluded from consideration by adding this clause to the clause sets \( \Psi_i \).

The identification and generalization of counterexamples to induction (CTIs) is a crucial step of the PDR method. For this reason, several variants of this step have already been considered: An effective algorithm based on exploring the lattice of subclauses of the given clause for computing minimal inductive subclauses has been proposed in [7]. The original implementation of the PDR method [6] used this algorithm for inductive generalization. A couple of changes of the original PDR method were proposed in [8], including an optimization for the cube generalization beyond non-inductive regions which has been integrated into the ABC tool [9]. Another clause generalization procedure implemented in the IIMV tool\(^1\) improves the performance of the PDR method significantly [5]: Instead of extracting a relative inductive clause from a CTI, the proposed procedure, called counterexamples to generalization (CTGs), also extracts inductive clauses from other states that are generated from failed attempts on generalizing some CTIs. Finally, [10] implemented different variants of the PDR method for hardware model checking in nuXmv\(^2\), and conducted a systematic evaluation using the benchmarks of the latest hardware model checking competition.

\(^1\)See http://ecee.colorado.edu/~bradleya/iimc.
\(^2\)See https://nuxmv.fbk.eu.
The original PDR method operates on top of a SAT solver whose queries are composed of the clause sets of the reachable state approximations $\Psi_0, \ldots, \Psi_k$, the initial states, and the transition relation of the considered hardware circuit. Since hardware circuits are usually synthesized from more abstract high-level languages like synchronous languages, it is natural to adapt PDR also to these higher levels of abstraction. The synchronous model of computation [11, 12] is thereby a suitable candidate for such a more abstract PDR method since it still reflects the execution of hardware circuits: In every macro step, new input values are read, internal states are updated, and outputs are immediately computed. All assignments are moreover done synchronously, so that all variables have a unique value per macro step (which is the logical time). Synchronous languages typically have a formal semantics and lend themselves well for formal verification, in particular, many compilers already make use of model checking for various analyses.

It is therefore natural to apply PDR to the formal verification of synchronous programs. To that end, one has to deal with (1) data types other than booleans, and (2) with statements that define explicit control- and dataflow of the programs. Both are at first considered as additional problems, but they can also be used to make the PDR method more efficient (compared to its application at the level of generated hardware circuits). Having the latter point of view, this paper is about one idea to exploit the control-flow of synchronous languages to improve the performance of the PDR method.

One of the first attempts to check synchronous programs by the PDR method is presented in [13]. However, the main topic of [13] was the integration of PDR with predicate abstraction for software model checking: It thus focused more on data types than on the control-flow of synchronous programs. The synchronous language Lustre considered there is a synchronous dataflow language which does not have an explicit control-flow as other synchronous languages like Esterel [14–17] and Quartz [18]. Exploiting control-flow information for PDR from synchronous programs has – to the best of our knowledge – not yet been considered. However, PDR has already been applied to the verification of sequential software [19–21] where the control-flow has been used to guide the PDR method: In particular, [19] introduced an algorithm called TREE-IC3 where the analysis of software programs is enabled by unrolling the control-flow graph of a program to an abstract reachability tree. Closer to the work presented in this paper is [21] where a sequential program’s control-flow structure is represented as a control-flow automaton. The authors integrated this explicit representation of control-flow with a stronger form of relative inductiveness over dataflow in comparison to the TREE-IC3 method. The main algorithms compute the weakest precondition that overapproximates the CTI states by SMT solvers for software model checking which is different to the generalization of CTIs at the bit-level that we tackle in this paper. Furthermore, it is unclear how to extend the method proposed in [21] to concurrent programs like synchronous programs.

In this paper, we present a simple idea that enhances the PDR method by considering the control-flow of synchronous languages. Our extension helps PDR (1) to decide about the unreachability of states derived from counterexamples, and (2) to generalize them to clauses that cover as many unreachable states as possible. The generated clause is then added to the reachable state approximation sequence $\Psi_0, \ldots, \Psi_k$ to exclude these unreachable states from further consideration. We present our ideas using our Esterel-like language Quartz [18], but emphasize that it is applicable to any state-based language with a clear distinction between control- and dataflow.

As we will explain, the transition relation $T$ of such a synchronous program can be derived as a conjunction $T = T^c \land T^d$ of transition relations over the same set of states, one for the control flow $T^c$ and another one for the dataflow $T^d$, respectively. The reachability of a state $s'$ from a state $s$ in $T$ is equivalent to the reachability in both $T^c$ and $T^d$. Hence, if $s'$ is not reachable from $s$ in $T^c$ only, we can already conclude its unreachability in $T$ and thus can declare it as a CTI without considering the full transition relation $T$. The advantage is that the state transition system defined by the control-flow $T^c$ is much simpler to deal with (even though it has even more reachable states) since we can compute a usually small quotient in terms of the extended finite state machine of the synchronous program. We may also use traditional model checking approaches for that purpose since the control-flow system $T^c$ can be usually represented efficiently by means of BDDs. This way, we can add a first, less expensive test for checking the unreachability of a state. If that test should fail, we use the traditional PDR reachability checks that will generate further reachability queries that then (again) first ask for reachability in the control-flow transition system $T^c$ in every step.

Second, having determined that a state is unreachable in $T^c$, we can reduce it to its control-flow variables, and obtain this way a generalized clause that will exclude all states of the transition system that refer to the same program locations (but with different values of the data variables). Thus, we can avoid expensive clause generalizations that are required in PDR to narrow the over-approximations of the clause sets $\Psi_0, \ldots, \Psi_k$. The clauses generated this way may not be relatively inductive, but by inspection of the control-flow part $T^c$, we can directly decide about their unreachability which is sufficient for excluding these states. Also, the clauses may not be minimal, but since they are quickly generated this way, it is usually a good compromise. Alternatively, we could apply the usual methods to minimize them.

The outline of the paper is as follows: In the next section, we briefly review the PDR method. Section III introduces the synchronous language Quartz and the symbolic representations of Quartz programs including the control- and dataflow parts $T^c$ and $T^d$. Using the control-flow part $T^c$, a control-flow guided clause generation method is proposed in Section IV. We demonstrate how the method works for a Quartz program in Section V. The paper will be concluded in Section VI.
II. Property Directed Reachability (PDR)

Let $\mathcal{K} := (\mathcal{V}, \mathcal{I}, T)$ be a state transition system over a set of Boolean variables $\mathcal{V}$ where $\mathcal{I}$ and $T$ are two propositional formulas that describe the initial condition and the transition relation over $\mathcal{V}$, respectively. Hence, the states of $\mathcal{K}$ are associated with subsets of variables $s \subseteq \mathcal{V}$ which are furthermore associated with variable assignments in that those variables in $s$ are considered be true while those outside $s$ are considered to be false. Thus, initial states are states $s \subseteq \mathcal{V}$ that are associated with variable assignments that satisfy $\mathcal{I}$, and transitions $(s, s')$ are pairs of states that satisfy the formula $T$.

We use modal formulas $\Box \varphi$ (as used in $\mu$-calculus, see e.g. [22]) that hold in a state $s$ if and only if all its successor states satisfy property $\varphi$. A property $\varphi$ is called inductive if $\varphi \rightarrow \Box \varphi$ is valid, i.e., no transition starting in states satisfying $\varphi$ will leave this state set. Moreover, $\varphi$ is called inductive relative to $\psi$ if $\psi \land \varphi \rightarrow \Box \varphi$ is valid, i.e., all successor states of $\psi \land \varphi$ are contained in $\varphi$.

To prove that a property $\Phi$ holds on all reachable states of $\mathcal{K}$, the PDR method constructs a sequence of clause sets $\Psi_0, \ldots, \Psi_k$ that include the states that are reachable in $0, \ldots, k$ steps, respectively. PDR first checks whether an initial state or a successor of the initial states violates the property $\Phi$. If so, a counterexample to $\Phi$ is already found. Otherwise, $\mathcal{I} \rightarrow \Phi \land \Box \Phi$ is valid, and PDR continues to check whether $\mathcal{I}$ or $\Phi$ are inductive. If so, we have a proof; otherwise, there are reachable states other than the initial states that satisfy $\Phi$ with successor states violating $\Phi$. Thus, the first $\Psi$-sequence for $k := 1$ is set up as $\Psi_0 := \mathcal{I}$; $\Psi_1 := \Phi$.

Having an approximation of the reachable state sequence $\Psi_0, \ldots, \Psi_k$, the main algorithm of PDR then checks whether $\Psi_k \rightarrow \Box \Phi$ is valid, i.e., whether there are some states in $\Psi_k$ that have successors violating $\Phi$. Depending on this, one of the following is done:

- **Propagation:** If $\Psi_k \rightarrow \Box \Phi$ is valid, then the PDR method adds $\Phi$ as a new clause set $\Psi_{k+1}$ after propagating clauses from $\Psi_k$ to $\Psi_{k+1}$ to narrow the over-approximation.

- **Blocking:** If $\Psi_k \rightarrow \Box \Phi$ is not valid, then the SAT solver generates a counterexample that satisfies $\Psi_k \land \neg \Box \Phi$, i.e., a (partial) variable assignment over $\mathcal{V}$, which is a cube $C_k$, i.e., a conjunction of literals of (some of) these variables. PDR now has to check whether one of these states is reachable from the initial states. If one of them is reachable, it is a real counterexample since the property $\Phi$ then does not hold in all reachable states. Otherwise, the states in $C_k$ are all unreachable, thus form a CFI that can now be removed from all predicates $\Psi_i$. To that end, PDR generalizes the cube $C_k$ by removing successively literals from the cube so that as many unreachable states as possible are removed.

The main algorithm therefore stops with either a counterexample, or with a proof as soon as one $\Psi_i$ satisfies $\Psi_i \equiv \Psi_{i+1}$.

The generation and generalization of new clauses is thereby a crucial step that influences the performance of the PDR method. In general, checking reachability of a cube $C_k$ and generalizing it is done by PDR as follows:

- **Reachability of Cubes:** To prove the reachability of (all states of) a cube $C_k$ within $k$ steps, we would essentially have to check whether $\Psi_{k-1} \rightarrow \Box C_k$ is valid. If it is then no state of $C_k$ can be reached within $k$ steps. Otherwise, there are states in $\Psi_{k-1}$ that have successor states in $C_k$ (but these could be unreachable).

Bradley observed that it suffices to check the validity of $\Psi_{k-1} \land \neg C_k \rightarrow \Box \neg C_k$ and $\mathcal{I} \rightarrow \neg C_k$ instead: If both are valid, we conclude that no state in $C_k$ is reachable within $k$ steps\(^4\), and we can therefore remove all of these states from the predicates $\Psi_0, \ldots, \Psi_k$. On the other hand, if $\mathcal{I} \rightarrow \neg C_k$ should not be valid, we see that some states of $C_k$ are initial states and are therefore reachable. Finally, if $\Psi_{k-1} \land \neg C_k \rightarrow \Box \neg C_k$ should not be valid, then there is another counterexample, thus another cube $C_{k-1}$, which contains states that satisfy $\Psi_{k-1} \land \neg C_k$ and that have at least one successor state in $C_k$. Therefore, we have to recursively check the reachability of $C_{k-1}$ in $k-1$ steps in the same way. Finally, if we get a counterexample $C_0$ from $\Psi_0 \land \neg C_1 \rightarrow \Box \neg C_1$, then the sequence $C_0, \ldots, C_k$ contains a path in the transition system reaching a state in $C_k$. Otherwise, we conclude that no state in $C_k$ is reachable in $k$ steps and can remove it from the approximations $\Psi_i$.

- **Generalization of Clauses:** Once a cube $C_k$ has been proved unreachable within $k$ steps, we may add its clause $\neg C_k$ to all approximations $\Psi_0, \ldots, \Psi_k$. However, PDR tries to generalize the clause $\neg C_k$ before adding it by removing single literals from the clause as long as it remains unreachable. This way as many unreachable states as possible are removed. To this end, PDR constructs the subclause lattice $L_C := (2^{\mathcal{V}}, \subseteq \Psi_i)$ whose elements are the subclauses of $\neg C_k$ that are ordered by the subclause relative relation $\subseteq \Psi_i$, defined as follows: Two subclauses $C_1, C_2 \in 2^{\mathcal{V}}$ satisfy the relation $C_1 \subseteq \Psi_i C_2$ if $C_1 \subseteq C_2$ and $C_1 \land \Psi_i \rightarrow \Box \Psi_i C_2$ holds. For example, assume that the SAT solver returns $C_k := \{p_1 \land \neg p_2\}$ as a CFI over $\mathcal{V} = \{p_1, p_2\}$. We have $\neg C_k = \{\neg p_1, p_2\}$, and the subclause lattice is defined by $L_C = (\{\}, \{\neg p_1\}, \{p_2\}, \{-p_1, p_2\}, \Psi_i)$. Starting from the top element $\neg C_k$ of $2^\mathcal{V}$, which is inductive relatively to $\Psi_i$ as proved by the above reachability analysis, all the subclauses of $\neg C_k$ that are inductive relatively to $\Psi_i$ can be computed. Among these, PDR

\(^4\)Assume $X_i$ are the states that are reachable within no more than $i$ steps, i.e., $X_0 := \mathcal{I}$ and $X_{i+1} := X_i \cup \text{succ}_{C_i}(X_i)$. We can prove by induction that $X_i \cap C_k = \{\}$ holds for $i \leq k$: The induction base $X_0 \cap C_k = \{\}$ is equivalent to the validity of $\mathcal{I} \rightarrow \neg C_k$ that we assumed. For proving $X_{i+1} \cap C_k = \{\}$, we have to prove by the definition $X_{i+1} := X_i \cup \text{succ}_{C_i}(X_i)$ that $X_{i} \cap C_k = \{\}$ and $\text{succ}_{C_i}(X_i) \cap C_k = \{\}$. The former follows by induction hypothesis, and the latter is derived as follows: By induction hypothesis, we have $X_i \cap C_k = \{\}$, so that $X_i \rightarrow \Psi_i \land \neg C_k$ is valid (e.g., $\Psi_i$ holds by construction of $\Psi_i$). Since $\Psi_i \rightarrow \Psi_{i-1}$ holds for $i < k$, also $X_i \rightarrow \Psi_{i-1} \land \neg C_k$ is valid, and thus, we finally conclude with $\Psi_{i-1} \land \neg C_k \rightarrow \Box \neg C_k$ that $X_i \rightarrow \Box \neg C_k$ holds, which finally implies $\text{succ}_{C_i}(X_i) \cap C_k = \{\}$.\(^5\)We do not explain these details of PDR in this paper, and refer to [1-5] for further details.
chooses one that does not contain any strict subclause that is inductive relatively to \( \Psi_i \) as a minimal inductive subclause. Instead of strengthening each \( \Psi_i \) with \( \neg C_k \), \( \Psi_i \) is then updated by \( \Psi_i \wedge c \), so that \( C_k \) and many other unreachable states are removed from \( \Psi_i \).

As can been, PDR generates many queries to a SAT/SMT solver for checking the reachability of a cube that has been derived from a counterexample. The generalization step also requires a substantial effort in that further queries to a SAT/SMT solver are generated. As we will show, our method aims at replacing these steps with a simpler first check that just considers the control-flow of a synchronous program. If that check is successful, we can directly derive a generalized clause to be added to the reachable state approximations.

III. SYNCHRONOUS LANGUAGES

In this section, we introduce the synchronous language Quartz and the symbolic representations of Quartz programs.

A. The Quartz Language

Quartz is a synchronous language that is derived from the Esterel language [14–17]. The core of the Quartz language is based on the synchronous reactive model of computation: The execution of a synchronous reactive system proceeds in discrete reaction steps where in each step, inputs were read from the environment, outputs are immediately generated as reaction to these inputs, and the internal state of the system is updated for the next reaction step. Reaction steps are declared in Quartz programs by pause statements (see Fig. 2). Each pause statement introduces a control-flow label that is given a unique name by the compiler or the programmer. All assignments between such pause statements are executed in virtually zero time. Assignments can be either immediate \( x = \tau \) or delayed \( \text{next}(x) = \tau \) which is similar to hardware circuits where the former describes a wire and the latter a register assignment.

Sophisticated statements like various forms of loops, triggered actions, preemption and suspension statements allow one to conveniently express complex reactive behaviors. For the complete overview of the language, one can refer to [18] for more details. We just list some of the statements used in this paper and give an idea of their meaning:

- \( x = \tau \) and \( \text{next}(x) = \tau \) (immediate/delayed assignment)
- \( \ell \) : pause (start/end of macro step)
- \( S_1 ; S_2 \) (sequence)
- \( S_1 \parallel S_2 \) (parallel statement)
- if(\( \sigma \)) \( S_1 \) else \( S_2 \) (conditional)
- while(\( \sigma \)) \( S \) (data-dependent loop)
- loop \( S \) (infinite loop)

As an example for a Quartz program, consider module ITELoop shown in Fig. 2a: \( i \) is a local array with \( n \) boolean variables. The initial value of \( i[0] \) is true, the other array elements are false (default initialization). The body statement of the module ITELoop is a conditional statement: if \( \neg i[0] \) holds, then the loop statement will be immediately started. In the second macro step inside the loop statement \( i[0] \) is assigned to false. It is not difficult to see that \( i[0] \) always holds in module ITELoop since the loop is not reachable.

B. Symbolic Representation of Quartz Programs

The Averest\textsuperscript{5} compiler computes for a given Quartz program \( S \) an equivalent set of guarded actions \( G_S \), i.e., pairs \((\gamma, \alpha)\) consisting of a trigger condition \( \gamma \) and an atomic action \( \alpha \). Actions are thereby immediate assignments \( x = \tau \) or delayed assignments \( \text{next}(x) = \tau \) where \( \gamma \) and \( \tau \) are program expressions. The compiler generates guarded actions for the local and output variables (for the dataflow), but also for the control-flow labels \( w \) of the \( w \) : pause statements that will define the control-flow.

For the following, we briefly explain the construction of symbolic representations for state transition systems which will be sufficient for this paper. To that end, assume that our intermediate representation by guarded actions contains the following immediate and delayed actions for some variable \( x \): \((\gamma_1, x = \tau_1), \ldots, (\gamma_p, x = \tau_p)\) \((\delta_1, \text{next}(x) = \pi_1), \ldots, (\delta_q, \text{next}(x) = \pi_q)\)

Figure 1 shows then the definition of the initial states predicate Init, and the transition relation Trans\( x \) for variable \( x \) which are explained as follows: The initial value of a variable \( x \) can only be determined by its immediate actions. Hence, if one of the guards \( \gamma_i \) of the immediate actions holds, the corresponding immediate assignment defines the value of \( x \). If none of the guards \( \gamma_i \) should hold, i.e., \( \Gamma := \bigvee_{j=1}^{p} \gamma_j \) is false, the initial value of \( x \) is determined by its default value.

The transition relation can be explained similarly: First, also the immediate assignments have to be respected for the current

\textsuperscript{5}See http://www.averest.org.
point of time, i.e., whenever a guard \( \gamma_i \) of the immediate actions holds, the corresponding immediate assignment defines the current value of \( x \). If one of the guards \( \delta_i \) of the delayed assignments holds at the current point of time, the next value of \( x \) is determined by the corresponding delayed assignment. Finally, if the next value of \( x \) is not determined by an action, i.e., neither \( \Gamma := \bigvee_{j=1}^p \gamma_j \) holds at the next point of time nor does \( \Delta := \bigvee_{j=1}^p \delta_j \) hold at the current point of time, then the next value of \( x \) is determined by the reaction to absence. Depending on whether it is a memorized or event variable, it may store its previous value or will be reset to a default value.

The definitions given in Figure 1 can be literally used to define input files for symbolic model checkers. For the remainder of the paper, it is important to distinguish between the control-flow and the data flow which are defined as follows where \( V_{df} \) denotes all names of pause locations (i.e., control-flow variables) and \( V_{cf} \) denotes all dataflow (input, output, and local) variables:

- \( I_{cf} := \bigwedge_{x \in V_{cf}} \text{init}_x \) and \( T_{cf} := \bigwedge_{x \in V_{cf}} \text{Trans}_x \)
- \( I_{df} := \bigwedge_{x \in V_{df}} \text{init}_x \) and \( T_{df} := \bigwedge_{x \in V_{df}} \text{Trans}_x \)
- \( I := I_{cf} \land T_{cf} \land I_{df} \land T_{df} \)

Note that the transition systems \( K := (V, I, T) \), \( K_{cf} := (V, I_{cf}, T_{cf}) \), and \( K_{df} := (V, I_{df}, T_{df}) \) are defined over the same states which are subsets of \( V := V_{cf} \cup V_{df} \), but have different transitions and initial states.

**Lemma 1** (Transition Systems of a Synchronous Program). For states \( s, s' \) of the transition systems \( K := (V, I, T) \), \( K_{cf} := (V, I_{cf}, T_{cf}) \), and \( K_{df} := (V, I_{df}, T_{df}) \) the following holds:

- There is a transition \( s \to s' \) in \( K \) iff this transition exists both in \( K_{cf} \) and \( K_{df} \).
- State \( s' \) can be reached from state \( s \), i.e., \( s \to^* s' \) in \( K \) iff it can be reached from \( s \) both in \( K_{cf} \) and \( K_{df} \).
- If \( s \to^* s' \) in \( K_{cf} \), then we also have \( s \to^* s'' \) in \( K_{df} \) for every state \( s'' \) with \( s' \cap V_{cf} = s'' \cap V_{cf} \).

The proof of the first proposition is straightforward since \( K \) is the synchronous product of the transition systems \( K_{cf} \) and \( K_{df} \). \( K := K_{cf} \times K_{df} \). The second one is proved by induction using the first proposition. For the third proposition, note that \( T_{cf} \) has been constructed by guarded actions that do not constrain the next values of the dataflow variables; only the values of the control-flow labels in state \( s' \) are determined by the values of the variables in state \( s \), while the dataflow variables are completely unconstrained.

Consider module ITELoop with \( N := 1 \). The symbolic representations of its transition systems are as follows:\footnote{We modified the formulas a little bit to increase readability. Moreover, we note that the compiler always introduces an additional control-flow label run that is initially false and true otherwise.}

- \( V_{cf} := \{p_1, p_2, \text{run}\} \)
- \( V_{df} := \{i[0]\} \)
- \( T_{cf} := \neg(p_1 \lor p_2 \lor \text{run}) \land \text{next}(\text{run}) \land \text{next}(p_1) \land (\text{next}(p_1) \rightarrow (\neg\text{run} \land \neg i[0]) \land \text{next}(p_2) \rightarrow \neg i[0])) \)
- \( T_{df} := i[0] \)
- \( T := (p_1 \rightarrow i[0]) \land (\neg\text{next}(p_1) \rightarrow (\text{next}(i[0]) \leftrightarrow i[0])) \)

The corresponding state transition diagrams are shown in Fig. 2c and Fig. 2d.

**C. Extended Finite State Machines (EFSMs)**

The Averest system can also generate extended finite state machines (EFSMs) for Quartz programs, see e.g., Fig. 2b. The EFSM is related with the transition system \( K := (V, I, T) \) for the control-flow, but contains for every subset of \( V_{cf} \) at most one node that is furthermore endowed also with the guarded actions of the dataflow.

The EFSM of a Quartz program has even in case of infinite data types only finitely many nodes and transitions. The EFSM has one node for every subset of \( V_{cf} \) of the control-flow labels. Each node \( s_i \) in the EFSM carries the following information:

- \( \text{Label}(s_i) \): the set of control-flow labels that hold in \( s_i \).
- \( \text{Exec}(s_i) \): the set of guarded actions that can be enabled in node \( s_i \).

Taking node \( \text{state}(1) \) in Fig. 2b as an example, it represents the control-flow state where control-flow labels \( p_1 \) and \( \text{run} \) are true, while \( p_2 \) is false, and the immediate assignment \( i[0] = \text{false} \) is encoded by a guarded action in the node \( \text{state}(1) \).

Moreover, transitions between control-flow states are labeled with path conditions that must hold to activate the transitions, and every transition corresponds to a macro step of the synchronous program. For any pair of nodes \( (s_i, s_j) \), there is a path condition \( \varphi(s_i, s_j) \) that must hold to take the transition. As shown in Fig. 2b, the transition from \( \text{state}(0) \) to \( \text{state}(1) \) can take place iff path condition \( \neg i[0] \) holds.

Typically the EFSMs are generated by a symbolic execution of the program using the transition rules of the operational semantics of the Quartz language. However, there is also an obvious relationship between the control-flow state transition system. To see this, we ignore the labels of the edges in the EFSM for defining reachability:

**Lemma 2** (Unreachability Checking by EFSMs): If a node with a control-flow label \( s \subseteq V_{cf} \) is not reachable in the EFSM from the initial node, then neither this state nor any other state \( s' \subseteq V_{cf} \cup V_{df} \) with \( s = s' \cap V_{cf} \) is reachable in \( K \), and thus also none of these states is reachable in \( K \).

Note however that the EFSM may contain infeasible paths like the edge between states \( \text{state}(0) \) and \( \text{state}(1) \) in Figure 2b. Since we ignore the labels of the edges for defining reachability in the EFSM, such nodes are reachable in the EFSM, but their corresponding states in \( K \) may be not reachable.
IV. Control-Flow Guided Clause Generation

In this section, we describe our method that can automatically generate unreachable clauses based on the analysis of the control-flow transition system of the synchronous programs. As we will show, the unreachability of some CTIs in \( \mathcal{K} \) can be proved by proving unreachability in the control-flow transition system \( \mathcal{K}^{cf} \) of the synchronous program. Dropping the dataflow variables in the cube will moreover generate an unreachable cube whose clause can be used to narrow the reachable state approximations maintained by PDR. Our extension of PDR is based on the following theorem:

**Theorem 1** (Control-flow Guided Clause Generation). Let \( P \) be a synchronous program with the transition systems \( \mathcal{K} = (V, I, T) \), \( \mathcal{K}^{df} = (V, I^{df}, T^{df}) \), and \( \mathcal{K}^{cf} = (V, I^{cf}, T^{cf}) \) as introduced in the previous section and let \( C \) be a cube over the variables \( V \).

- **Reachability of Cubes:** If no state \( s \subseteq V \) that satisfies \( C \) is reachable in \( \mathcal{K}^{cf} \), then none of these states is reachable in \( \mathcal{K} \).
- **Generalization of Clauses:** If no state of \( C \) is reachable in \( \mathcal{K}^{cf} \), then also no state of \( C' := C|_V^{df} \) is reachable neither in \( \mathcal{K}^{cf} \) nor in \( \mathcal{K} \) where \( C' := C|_V^{df} \) denotes the restriction of \( C \) to the control-flow variables \( V^{cf} \).

**Proof.**

- The first proposition follows almost directly from Lemma 1, since unreachability in \( \mathcal{K}^{cf} \) implies unreachability in \( \mathcal{K} \).
- The cube \( C' \) contains only control-flow literals. One of its states is reachable in \( \mathcal{K}^{cf} \) iff the corresponding state endowed with the dataflow literals in \( C \) will be reachable in \( \mathcal{K}^{cf} \), since reachability in \( \mathcal{K}^{cf} \) does not depend on the dataflow literals. Thus, if no state of \( C \) is reachable in \( \mathcal{K}^{cf} \) then and only then, no state of \( C' \) is reachable in \( \mathcal{K}^{cf} \), and thus also not reachable in \( \mathcal{K} \).

The subclause \( C' := C|_V^{df} \) obtained from omitting the dataflow literals in cube \( C \) can therefore be used to narrow the reachable state approximations of PDR if no state of \( C \) is reachable in \( \mathcal{K}^{cf} \). Finally, checking unreachability of a cube \( C \) in \( \mathcal{K}^{cf} \) can be approximated by checking whether the node in the EFSM that corresponds with \( C' := C|_V^{df} \) is reachable in the EFSM.

As shown in Fig. 3, the Averest compiler computes for a given Quartz program an equivalent set of guarded actions \( \mathcal{G}_p \). Different transformation procedures are provided to adapt the guarded actions for special needs. The symbolic transition system \( \mathcal{K} \) is the input for PDR method. In the blocking phase, if the CTI can not be mapped to any of the reachable nodes of the EFSM, then an unreachable subclause will be derived by restricting the cube clause to its control-flow literals, otherwise the reachability checks of PDR will be applied.
V. EXAMPLE

In this section, we demonstrate our control-flow guided clause generation method by proving that \( i[0] \) holds on all reachable states of the transition system \( K \) of module ITELoop with \( N := 1 \) defined in Section III-B.

Fig. 2c and Fig. 2d show the transition systems of module ITELoop with \( N := 1 \). The yellow region covers the reachable states, and the green nodes are the states satisfying \( i[0] \), while the remaining ones violate \( i[0] \). Module ITELoop has in total 8 reachable control-flow states in \( K_{cf} \) which correspond to 4 nodes of the EFSM shown in Fig. 2b. We prove that \( \Phi := i[0] \) holds on all reachable states of \( K \) as follows:

- \( i[0] \) holds in both the initial state \( s_1 \) and its successor \( s_3 \). Since the successor \( s_3 \) of initial state \( s_1 \) is no initial state, the initial states are not inductive. Since there are transitions from states where \( \Phi := i[0] \) holds to states where \( \Phi \) does not hold, also \( \Phi \) is not inductive. So, we set up the first \( \Psi \)-sequence of clauses sets for \( k := 1 \):
  \[
  \Psi_0 := \{i[0], \neg p1, \neg p2, \neg \text{run}\}
  \]
  \[
  \Psi_1 := \{i[0]\}
  \]

- Let \( [\Psi_i]_K \) represent the set of states satisfying \( \Psi_i \). Looking at Fig. 2d, we see the following:
  \[
  [\Psi_0]_K := \{s_1\}
  \]
  \[
  [\Psi_1]_K := \{s_1, s_3, s_5, s_7, s_9, s_{11}, s_{13}, s_{15}\}
  \]
  \( s_5 \) and \( s_7 \) are CTIs since these states belong to \( [\Psi_1]_K \) but have successors violating \( i[0] \).
- We choose the CTI with the smallest index, i.e., \( s_5 \). It can be represented as cube \( \neg p1 \land p2 \land \neg \text{run} \land i[0] \), whose control-flow part is \( \neg p1 \land p2 \land \neg \text{run} \) which corresponds to states \( s_4, s_5 \). There is no corresponding node in the EFSM labeled with \( \{p2\} \), and thus, none of the states \( s_4, s_5 \) are reachable in \( K_{cf} \), and thus neither in \( K \). Therefore, unreachability of \( s_4, s_5 \) follows directly, and we could add clause \( \{p1, \neg p2, \text{run}\} \) to \( \Psi_0 \) and \( \Psi_1 \).
- To generalize the clause, we next explore the lattice of the subclauses of \( \{p1, \neg p2, \text{run}\} \). The minimal inductive subclause \( \{\neg p2\} \) can be extracted. We then conjoin \( \{\neg p2\} \) and obtain the following \( \Psi \)-sequence:
  \[
  \Psi_0 := \{i[0], \neg p1, \neg p2, \neg \text{run}\}
  \]
  \[
  \Psi_1 := \{i[0], \neg p2\}
  \]

We now have \( [\Psi_1]_K = \{s_1, s_3, s_9, s_{11}\} \).
- Since now all successors of \( \Psi_1 \) satisfy \( i[0] \), PDR increments the trace and propagates clauses as usual in PDR. The following \( \Psi \)-sequence of clause sets for \( k := 2 \) is obtained:
  \[
  \Psi_0 := \{i[0], \neg p1, \neg p2, \neg \text{run}\}
  \]
  \[
  \Psi_1 := \{i[0], \neg p2\}
  \]
  \[
  \Psi_2 := \{i[0], \neg p2\}
  \]

Note that \( \{\neg \text{run}\} \) cannot be propagated to \( \Psi_1 \) since it does not hold on \( s_3 \), but \( \neg p1 \) propagates from \( \Psi_0 \) to \( \Psi_1 \) since it holds on all successors of \( \Psi_0 \), i.e., on \( s_3 \).
- We now have \( \Psi_1 \equiv \Psi_2 \) (syntactic equality) with \( [\Psi_1]_K = [\Psi_2]_K = \{s_1, s_3\} \), so we found a proof and conclude that \( i[0] \) holds on all reachable states of \( K \).

In general, the set of boolean variables of module ITELoop is the following:

\[
\gamma^\text{cf} := \{i[0], \ldots, i[N-1]\} \cup \{p1, p2, \text{run}\}
\]

In the worst case, \( C \) contains \( N + 3 \) literals over \( \gamma^\text{cf} \). Omitting the dataflow literals, only three literals over \( \gamma^\text{cf} \) remain. By our control-flow guided clause generalization method, starting from those CTIs that cannot be mapped to any nodes in the EFSM, the PDR method benefits from the following two aspects:

- The unreachability of those CTIs can be proved directly by the control-flow states in the EFSM.
- It is sufficient to narrow the reachable state approximations with the generalized clause \( C' \) obtained from omitting the dataflow literals. In this way, the traditional generalization of clauses, which may yield \( 2^{k+3} \) queries to a SAT/SMT solver, can be avoided.

- The generalized clause \( C' \) may not be relative inductive. If it is relative inductive, then deriving a minimal inductive subclause from it can exclude more unreachable states, which requires at most \( 2^k \) times relative inductiveness reasoning.

Our control-flow guided method can safely omit dataflow literals for clause generation so that the more expensive clause generalization of PDR will only be called when it is really needed. The generalized clause generated by our method excludes all states of the transition system that refer to the same program locations.

\footnote{A clause \( c \) of \( \Psi_i \) is propagated to \( \Psi_{i+1} \) if \( \Psi_i \rightarrow \Box c \) is valid.}
VI. Conclusions

PDR constructs a sequence of clause sets that over-approximate the states that are reachable in finitely many steps. The generalization of clauses that cover unreachable states is a crucial step that influences the performance of the blocking phase of the PDR method. In this paper, we described a method that can automatically generate clauses based on the analysis of the control-flow of synchronous programs. The unreachable of some cubes can be directly proved by only inspecting the unreachable control-flow states of the synchronous program. Omitting the dataflow literals of a counterexample’s cube yields a generalized clause in these cases.

References